Problem 1: Symmetric Binary Tree

Code

```
#include <iostream>
#include <algorithm>
using namespace std;

struct Node {
  Node* left{};
  Node* right{};
  int val{};
  Node() {
    left = nullptr;
    right = nullptr;
    val = 0;
  }
  explicit Node(int val) {
    left = nullptr;
    right = nullptr;
    this->val = val;
  }
};

int symmetricCount(Node* left, Node* right) {
  if (left == nullptr && right == nullptr)
    return 0;
  if ((left == nullptr && right != nullptr) || (left != nullptr &&
right == nullptr) || left->val != right->val)
    return -1;
  int l = symmetricCount(left->left, right->right);
  if (l == -1)
    return -1;
  int r = symmetricCount(left->right, right->left);
  if (r == -1)
    return -1;
  return l + r + 2;
```

```
}

int main() {
  int size;
  cin >> size;
  Node nodeArray[size];
  for (int i = 0; i < size; i++) {
    int num{};
    cin >> num;
    nodeArray[i] = Node(num);
  }
  for (int i = 0; i < size; i++) {
    int left{}, right{};
    cin >> left >> right;
    if (left != -1)
      nodeArray[i].left = &nodeArray[left - 1];
    if (right != -1)
      nodeArray[i].right = &nodeArray[right - 1];
  }
  int res = 1;
  for (auto p : nodeArray) {
    res = max(res, symmetricCount(p.left, p.right) + 1);
  }
  cout << res << endl;
  return 0;
}
```

What does my algorithm do in the program:

1. Initialize all the nodes and put them ordinarily into the array without assigning their children pointers.
2. Iterate the array to assign the children pointers for each node. Then we get a tree structure.
3. Design a function `symmetricCount` to check if it is a symmetric binary tree given the root. We need to check whether its left-subtree and right-subtree mirror each other. If yes, return the number of nodes of these two sub-trees. If no, return -1. We can apply this trick recursively to make a `DFS` check.

Why do I choose this method:

There is a similar exercise on Leetcode. But it just requires giving out a `boolean` value. We can modify the solution to fit this question. That is, use some impossible values of numbers of nodes to denote the asymmetric case and take advantage of recursion to make a count of nodes. I prefer `DFS` because recursion is much easier to implement than iteration, where only I can use `DFS`. But also, `DFS` saves more spaces, as `BFS` needs an extra queue.

Problem 2: Non-repeating Numbers

Code

```
#include <iostream>
#include <unordered_set>
using namespace std;

int main() {
  int dataset;
  cin >> dataset;
  for (int i = 0; i < dataset; i++) {
    int size;
    cin >> size;
    unordered_set<int> hashSet(50001);
    for (int j = 0; j < size; j++) {
      int num;
      cin >> num;
      if (hashSet.find(num) == hashSet.end()) {
        printf("%d ", num);
        hashSet.insert(num);
      }
    }
    printf("\n");
  }
  return 0;
}
```

What:

Use the hash map to store iterated elements. If the new one does not exist in the hash map, keep it and print it.

Why:

1. We need a data structure to store iterated elements to classify redundant ones.
2. We want to make the access of each element extremely fast because we need to search for the new come in one every time.

Time Complexity and Why:

Expectedly O(n). Hash one element and put it into the hash map, O(1). Checking one element if it exists in the hash map takes O(1). And we need to handle n elements. So, O(n) (Do not consider rehash).

Problem 3: Wandering

Code

```
#include <cstdio>
#include <climits>
#include <queue>
#include <vector>
using namespace std;

struct lengthNodes {
  int length;
  int nodeIndex;
  lengthNodes(int length, int nodeIndex) {
    this->length = length;
    this->nodeIndex = nodeIndex;
  }
};

vector<int> dijkstra(int src, const vector<vector<pair<int,
int>>>& graph) {
  vector<int> distance(graph.size(), INT_MAX);
```

```
  using my_value = lengthNodes;
  using my_container = vector<my_value>;
  auto my_comp = [](const my_value& t1, const my_value& t2) {
    return t1.length > t2.length;
  };
  priority_queue<my_value, my_container, decltype(my_comp)>
priorityQueue{my_comp};
  priorityQueue.push(lengthNodes(0, src));
  while (!priorityQueue.empty()) {
    while (distance[priorityQueue.top().nodeIndex] != INT_MAX) {
      priorityQueue.pop();
      if (priorityQueue.empty())
        return distance;
    }
    auto top = priorityQueue.top();
    priorityQueue.pop();
    distance[top.nodeIndex] = top.length;
    for (const auto& p : graph[top.nodeIndex]) {
      int newDistance = p.second + top.length;
      priorityQueue.push(lengthNodes(newDistance, p.first));
    }
  }
  return distance;
}

int main() {
  int verticesNums{}, edgeNums{};
  scanf("%d %d\n", &verticesNums, &edgeNums);
  vector<vector<pair<int, int>>> graph(verticesNums);
  for (int i = 0; i < edgeNums; i++) {
    int v1{}, v2{}, length{};
    scanf("%d %d %d\n", &v1, &v2, &length);
    graph[v1 - 1].push_back({v2 - 1, length});
    graph[v2 - 1].push_back({v1 - 1, length});
  }
  auto zeroToAll = dijkstra(0, graph);
  auto lastToAll = dijkstra(verticesNums - 1, graph);
  unsigned int res = INT_MAX;
  int smallest = lastToAll[0];
  for (int i = 0; i < verticesNums; ++i) {
    for (auto p : graph[i]) {
```

```
      unsigned int sum = zeroToAll[i] + lastToAll[p.first] +
p.second;
      if (sum != smallest) {
        res = min(res, sum);
      }
    }
  }
  printf("%d", res);
  return 0;
}
```

What:

1. Use the Dijkstra algorithm with a priority queue.
2. Use the adjacent list to store the graph.
3. The comparably shorter paths between `SOURCE` and `END` can be represented as `SOURCE TO V` + `V TO W` + `W TO END`, where `V` and `W` are intermediate vertices. Among all these paths, the shortest ones must be the shortest path. And the one just a little bit longer is the second shortest path.
4. Iterate `V` and `W` throughout all vertices; we can always find the second shortest one slightly larger than the shortest one.

Why:

1. I guess the time limit does not allow other algorithms that we have learned. 😂
2. We need to design a method to determine the distance from one source to all nodes. The Dijkstra algorithm perfectly fits this requirement.

Problem 4:

Code

```
#include <cstdio>
#include <vector>
#include <queue>
#include <cmath>
using namespace std;
```

```
struct lengthNode {
  unsigned long distance;
  pair<int, int> nodes;
  lengthNode(unsigned long distance, pair<int, int>& nodes) {
    this->distance = distance;
    this->nodes = nodes;
  }
};

void mergeAintoB(int indexAval, int indexBval, vector<unsigned
int>& status) {
  for (auto& n : status) {
    if (n == indexAval)
      n = indexBval;
  }
}

int main() {
  int verticesNums, leastLength;
  scanf("%d %d", &verticesNums, &leastLength);
  vector<pair<int, int>> coordinates{};
  for (int i = 0; i < verticesNums; ++i) {
    int x, y;
    scanf("%d %d", &x, &y);
    coordinates.emplace_back(x, y);
  }
  using my_valuetype = lengthNode;
  using my_container = vector<my_valuetype>;
  auto my_comp = [](const my_valuetype& p1, const my_valuetype&
p2) { return p1.distance > p2.distance; };
  priority_queue<my_valuetype, my_container, decltype(my_comp)>
priorityQueue{my_comp};
  for (int i = 0; i < verticesNums - 1; i++) {
    for (int j = i + 1; j < verticesNums; j++) {
      long euclidDis =
          pow(coordinates[i].first - coordinates[j].first, 2) +
pow(coordinates[i].second - coordinates[j].second, 2);
      if (euclidDis >= leastLength) {
        auto k = pair<int, int>(i, j);
        priorityQueue.push(lengthNode(euclidDis, k));
```

```
      }
    }
  }
  vector<unsigned int> status(verticesNums, 0);
  auto k = priorityQueue.top();
  priorityQueue.pop();
  status[k.nodes.first] = 1;
  status[k.nodes.second] = 1;
  unsigned int nums = 2;
  unsigned long res = k.distance;
  while (!priorityQueue.empty()) {
    auto p = priorityQueue.top();
    priorityQueue.pop();
    if (status[p.nodes.first] == 0 && status[p.nodes.second] == 0)
{
      status[p.nodes.first] = nums;
      status[p.nodes.second] = nums++;
      res += p.distance;
    } else if (status[p.nodes.first] == 0 &&
status[p.nodes.second] != 0) {
      status[p.nodes.first] = status[p.nodes.second];
      res += p.distance;
    } else if (status[p.nodes.first] != 0 &&
status[p.nodes.second] == 0) {
      status[p.nodes.second] = status[p.nodes.first];
      res += p.distance;
    } else if (status[p.nodes.second] != status[p.nodes.first]) {
      mergeAintoB(max(status[p.nodes.first],
status[p.nodes.second]),
                  min(status[p.nodes.first],
status[p.nodes.second]),
                  status);
      res += p.distance;
    }
  }
  for (auto n : status) {
    if (n != 1) {
      printf("%d", -1);
      return 0;
    }
  }
```

```
  printf("%lu", res);
  return 0;
};
```

What:

1. Store all nodes' positions.
2. Calculate all possible edges distance. And put valid ones into the priority queue.
3. Use Kruskal's Algorithm

Why:

The question is actually asking whether we could find a minimum spanning tree to make it a connected graph. We just need to modify Kruskal's Algorithm a little bit.

Time Complexity and Why:

O(n^2) (n represents the number of nodes). Calculating all possible edges takes two loops, which is O(n^2). During the Kruskal procedure, the alias of a node may need to be changed up to $n-1$ times. Consider we have n nodes. So, it is O(n^2).